A Library for the Solution of Sparse Systems on Distributed Memory Computers

Jacko Koster

The solution of large sparse linear systems lies at the heart of many calculations in computational science and engineering and is of increasing importance in computations in the financial and business sectors. Today, systems of equations with more than one million unknowns need to be solved. To solve such large systems in a reasonable time requires the use of powerful parallel computers. To date, only limited software for such systems has been available. The European project PARASOL aimed to design and develop a library of scalable sparse matrix solvers for distributed memory computers. The CLRC Rutherford Appleton Laboratory, CERFACS, and ENSEEIHT, were jointly responsible for the direct solvers. In this context, we have developed a MUltifrontal Massively Parallel Solver (MUMPS).

PARASOL is an ESPRIT IV Long Term Research Project for developing "An Integrated Environment for Parallel Sparse Matrix Solvers". The Project started in January 1996 and finished in June 1999. The Project involved twelve partners from five European countries, including leading academic groups with experience in the development of parallel solvers and industrial code developers who will integrate the PARASOL solvers into their software. An important aspect of the project was the strong link between the developers of the sparse solvers and the industrial end users who provided a range of test problems and evaluated the solvers. Figures 1 and 2 show two problems from the automotive industry that are modelled by MacNeal-Schwendler (Munich, Germany) as finite-element problems. The problems have 148,770 and 227,362 unknowns, respectively.

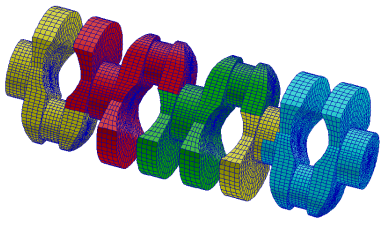

Figure 1: BMW crankshaft

To address a wide range of applications, the PARASOL library includes both state-of-the-art direct solvers and iterative solvers. The latter are based on domain decomposition techniques developed by Parallab (Bergen, Norway) and ONERA (Paris,

France), as well as multigrid techniques developed by GMD (Bonn, Germany). The CLRC Rutherford Appleton Laboratory (Chilton, England), CERFACS, and ENSEEIHT (both Toulouse, France), were jointly responsible for the direct solvers and developed the MUltifrontal Massively Parallel Solver, referred to as MUMPS [3, 4]. The final PARASOL library will be available in the public domain.

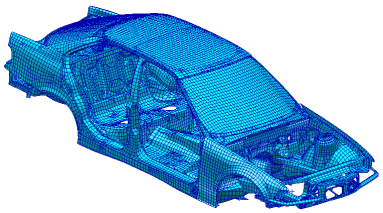

Figure 2: BMW carbody

MUMPS has been designed to solve a large range of sparse linear systems of the form $\mathbf{Ax} = \mathbf{b}$, Here, $\mathbf{A}$ is a symmetric positive definite, general symmetric, or unsymmetric matrix that is possibly rank deficient, $\mathbf{b}$ is the right-hand side vector, and $\mathbf{x}$ is the solution vector to be computed. MUMPS factorises an unsymmetric matrix $\mathbf{A}$ into two triangular matrices (the factors) $\mathbf{L}$ and $\mathbf{U}$ so that $\mathbf{LUx} = \mathbf{b}$. The vector $\mathbf{x}$ is obtained by first computing $\mathbf{y} = \mathbf{L}^{-1}\mathbf{b}$ (using forward substitution) and then $\mathbf{x} = \mathbf{U}^{-1}\mathbf{y}$ (using back substitution). The factorisation of the matrix $\mathbf{A}$ is in general the most expensive part of the solution process. MUMPS uses an LDL^T factorisation for symmetric matrices.

The main issue with sparse direct solvers is that the factors $\mathbf{L}$ and $\mathbf{U}$ will contain more nonzero entries than the original matrix, sometimes markedly so. (Zero entries are not stored.) This increase in nonzero entries is called fill-in. The amount of fill-in depends on both the nonzero pattern of the original matrix and the order in which the unknowns are eliminated by the solver. The ordering also largely determines the amount of work (floating-point operations) required during the factorisation. It is thus necessary that a direct solver orders the unknowns such that both time and storage requirements are minimized. Unfortunately, computing the optimal ordering is an NP-complete problem and in practice orderings are computed using heuristics.

The MUMPS package uses a multifrontal approach to factorise the matrix [1, 2]. The principal feature of a multifrontal method is that the overall factorisation is described (or driven) by an assembly tree (Figure 3). Each node in the tree represents some computation and each edge represents the transfer of data from a child node to its parent (which is the adjacent node in the direction of the root). An important aspect of the assembly tree is that it only defines a partial order for the factorisation. That is, arithmetic operations at a pair of nodes where neither is an ancestor of the other are independent. For example, work can commence in parallel on all the leaf nodes of the tree. Operations at the other nodes can proceed as soon as the data is available from the children of the node. There is thus good scope for exploiting parallelism, especially since assembly trees for practical problems contain many thousands of nodes.

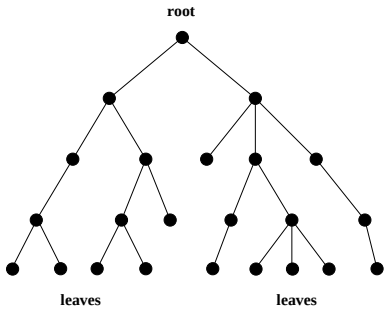

Figure 3: Example assembly tree

Another important feature of the multifrontal method is that the operations at each node in the assembly tree take place within a dense submatrix, called frontal matrix. The frontal matrix can be partitioned as a 2×2 block matrix

$$\begin{pmatrix} \mathbf{F}_{11} & \mathbf{F}_{12} \\ \mathbf{F}_{21} & \mathbf{F}_{22} \end{pmatrix}.$$

Unknowns are eliminated (using steps of Gaussian elimination) from within the block $\mathbf{F}_{11}$ only. The Schur complement matrix $\mathbf{F}_{22} - \mathbf{F}_{21}\mathbf{F}_{11}^{-1}\mathbf{F}_{12}$, called the contribution block, is computed and sent to the parent node in the assembly tree where it is assembled (or summed) into the corresponding frontal matrix. This requires indirect addressing. For the elimination operations on a frontal matrix and forming the contribution block, we can use high performance (cache-efficient) dense matrix kernels that are usually supplied by the machine vendors.

The parallel code solves the system $\mathbf{Ax} = \mathbf{b}$ in three main steps:

1. *Analysis.* A master processor computes an ordering of the unknowns based on the symmetrised matrix pattern $\mathbf{A} + \mathbf{A}^T$. The ordering can also be provided by the user. The master also computes the assembly tree and a mapping of the nodes of the tree to the processors. The master then sends this and other symbolic information to the other processors. Using this information, each processor estimates the work space required for its part of the factorisation and solution.

2. *Factorisation.* The original matrix is first preprocessed (if necessary) and distributed to the processors. Each processor allocates the memory for workspace and factors. The numerical factorisation on each frontal matrix is performed by a processor determined by the mapping and potentially one or more other processors that are determined dynamically. The factors must be kept for the solution phase.

3. *Solution.* The right-hand side **b** is broadcast from the master to the other processors. They compute the solution **x** by forward and back substitution using the distributed factors. The solution vector is then assembled on the master.

The MUMPS package thus contains three main entries. Repeated factorisations are possible using the symbolic information from one analysis, and repeated solutions are possible by re-using the factors of one factorisation. This way, some computationally expensive parts of the whole solution process need not be repeated in situations where a sequence of similar systems of equations need be solved (for example systems whose matrices have the same pattern or systems that only differ in the right-hand side).

The mapping that is computed during the analysis keeps communication costs to a minimum during factorisation and solution and balances the memory and computation required by the individual processors. The computational cost is approximated by the number of floating-point operations, assuming no pivoting is performed, and the storage cost by the number of entries in the factors. If two adjacent nodes in the tree are assigned to different processors, the connecting edge represents communication between the processors. Such communication is an expensive overhead. Therefore, MUMPS identifies subtrees in the assembly tree and maps a complete subtree onto a single processor of the target machine (Figure 4). In general, MUMPS uses more subtrees than there are processors. This way, the computation performed on the subtrees can be balanced over the processors.

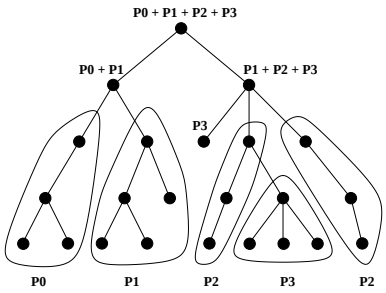

Figure 4: Example distribution of the computation over four processors **P0**, **P1**, **P2**, and **P3**

In practice, the nodes near the root of the assembly tree involve more computation than nodes further away from the root. For some problems we observed that more than 75% of the work is performed in the top three levels of the tree. Unfortunately, the number of independent nodes near the root is small, and so there is less parallelism to exploit. For example, the root in Figure 4 has only two children and hence, only two processors would perform the corresponding work while the other processors remain idle.

It is thus necessary to obtain further parallelism within the nodes near the root. In the MUMPS implementation, the processor to which a computationally expensive node is assigned, partitions the corresponding frontal matrix and maps the parts dynamically onto a number of processors that have a relatively low load. The processor also informs all the processors that are involved in the elimination operations on the children nodes so that they can send their data (contribution blocks, etc.) directly to the appropriate processors. The elimination operations

on the frontal matrix are subsequently performed in parallel.

Overall, MUMPS achieves high performance by exploiting parallelism due to the sparsity of the problem and parallelism from dense matrix kernels. Furthermore, MUMPS overlaps computation with communication by using asynchronous communication. MUMPS uses dynamic data structures and dynamic scheduling of computational tasks to accommodate extra fill-in in the factors due to numerical considerations (not taken into account during the analysis) and to cope with load variations of the processors.

So far, the software has mainly been used for solving problems from industrial partners in the project. Typical PARASOL test cases are from application areas such as computational fluid dynamics, structural mechanics, modelling compound devices, modelling ships and mobile offshore platforms, industrial processing of complex non-Newtonian liquids, and modelling car bodies and engine components. Table 1 shows the performance of the MUMPS factorisation and solution phases on a symmetric positive definite matrix (provided by MacNeal-Schwendler) that comes from the modelling of an inline skater.

number of processors	elapsed time	
	factorisation	solution
1	723	18.5
2	385	10.7
4	222	8.8
8	151	5.0
12	97	4.4
16	68	4.2
32	62	4.4

Table 1: Factorisation and solution time (in seconds) for MUMPS on the INLINE500K test case ($503,712$ unknowns) on an SGI Origin 2000 (195Mhz) machine.

The matrix is of order $503,712$ and has 36.8 million nonzeros in its lower triangular part. The factorisation requires 143 Gigaflops and the factors contain 175 million entries. The largest problem we have solved to date is a model of an AUDI crankshaft. The corresponding linear system is symmetric positive definite and of order $943,695$ with more than 39 million entries in its lower triangular part. With the best ordering of the unknowns that we tried, MUMPS created 1.4 billion entries in the factors and required 5.9 Teraflops for the factorisation of the matrix.

MUMPS is designed to be used in conjunction with other solution techniques and this will allow even larger problems to be solved. For example, within substructuring methods, the overall problem is decomposed into subdomains and the corresponding global linear system can be partitioned as

$$\begin{pmatrix} \mathbf{A}_{11} & & & \mathbf{A}_{1p} \\ & \mathbf{A}_{22} & & \mathbf{A}_{2p} \\ & & \ddots & \vdots \\ \mathbf{A}_{p1} & \mathbf{A}_{p2} & \dots & \mathbf{A}_{pp} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_p \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \vdots \\ \mathbf{b}_p \end{pmatrix}$$

Here, the submatrix $\mathbf{A}_{ii}$, $i < p$, corresponds to the (interior) unknowns in subdomain i, $\mathbf{A}_{pp}$ to the unknowns that lie on the interfaces between subdomains, and the other matrices represent the coupling between the subdomains and the interface. The subdomains are decoupled and can therefore be factorised independently (e.g., by MUMPS). The overall factorisation then reduces to factorising the interface, that is, the Schur complement matrix

$$\mathbf{S} = \mathbf{A}_{pp} - \sum_{i=1}^{p-1} \mathbf{S}_i \ , \quad \mathbf{S}_i = \mathbf{A}_{pi}\mathbf{A}_{ii}^{-1}\mathbf{A}_{ip}$$

where $\mathbf{S}_i$ is a local Schur complement matrix. However, the explicit construction of $\mathbf{S}$ is often expensive, in terms of both time and storage. To avoid this, iterative techniques are usually applied to the interface (that do not need $\mathbf{S}$ explicitly). For example, the domain decomposition solver developed by Parallab uses MUMPS on the subdomains and a preconditioned conjugate gradients iteration on the interface. To enhance the performance of the overall method, MUMPS has options to compute the rank and a null space basis of matrices (the subdomain matrices may be rank-deficient), and to return a Schur complement matrix.

The MUMPS software is also equipped with a range of classical pre- and postprocessing facilities that may speed up the computation and improve the computed solution. For example, MUMPS has several ways to scale the rows and columns of badly conditioned matrices prior to factorisation. The system solved is then

$$(\mathbf{D}_1\mathbf{A}\mathbf{D}_2)(\mathbf{D}_2^{-1}\mathbf{x}) = \mathbf{D}_1\mathbf{b}$$

where $\mathbf{D}_1$ and $\mathbf{D}_2$ are diagonal matrices. Matrix scaling is often useful to reduce the amount of numerical pivoting (which usually requires extra storage and work). MUMPS can also do a (backward) error analysis of the computed solution and perform iterative refinement to improve the accuracy of the solution. In situations where the user has good knowledge of the problem at hand and knows a good (or even optimal) order in which the unknowns should be eliminated, MUMPS can accept a user-defined (external) ordering of the unknowns, instead of computing one internally.

The MUMPS (and other PARASOL) software is written in Fortran 90. It requires MPI for message passing and makes use of BLAS, LAPACK, BLACS, and ScaLAPACK subroutines. MUMPS has been developed and tested on an IBM SP2, an SGI Power Challenge, and an SGI Origin 2000. The software is currently being ported to a Cray T3E.

Although the PARASOL project finished in June 1999, the MUMPS software is being further tested and developed. The main contributors to the MUMPS project are Patrick Amestoy (ENSEEIHT-IRIT), Iain Duff (RAL/CERFACS), Jean-Yves L'Excellent (CERFACS), Miroslav Tůma (Czech Academy of Sciences) and the author.

Further information on the PARASOL project is available from the Web URL

`http://www.genias.de/parasol/`

References

[1] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[2] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984.

[3] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. Technical Report RAL-TR-98-051, Rutherford Appleton Laboratory, Chilton, Didcot, England, 1998. To appear in a special issue of *Comput. Methods in Appl. Mech. Eng. on Domain Decomposition and Parallel Computing.*

[4] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. Technical Report RAL-TR-1999-059, Rutherford Appleton Laboratory, Chilton, Didcot, England, 1999.

About the author

Jacko Koster completed the OOTI course in 1992. The second year of this course he worked at the Koninklijke/Shell Laboratorium Amsterdam (Shell Research B.V.) where he started working on parallel linear algebra algorithms and implementations. After OOTI, he moved to CERFACS (Toulouse). In 1997, he received a PhD degree in Informatics at the Institut National Polytechnique de Toulouse. Afterwards, he moved to the CLRC Rutherford Appleton Laboratory (Chilton) where he worked in the Numerical Analysis Group of the Computational Science and Engineering Department. Since September 1999, he works at Parallab at the University of Bergen, Norway.